Efficient Broadcast on Computational Grids

Gabriel Mateescu  
IMSB  
National Research Council  
gabriel.mateescu@nrc.gc.ca

Ryan Taylor  
School of Computer Science  
Carleton University  
rtaylor@scs.carleton.ca

May 12, 2003
• MPI programs can contain point-to-point and collective communication operations
• Collective communication operations (broadcast, scatter, gather) are potential performance bottlenecks for scientific computing codes
• Efficient broadcast is needed for wide area and grid computing that uses collective communication
  – The penalty of inefficient global communication is higher on wide area networks than on clusters and local area networks
Problem Formulation

• Set of networked computer resources represented as a strongly connected graph \( G = (V,E) \)
  – Vertices represent computer nodes
  – Edges represent the interconnect
  – Each edge \((u,v)\) in \(E\) has a weight \(w(u,v)\): the latency of communication between \(u\) to \(v\)

• A message is sent from a designated root to all processes such that:
  – For each edge \((u,v)\), it takes time \(\Delta(u,v)\) to inject the message at \(u\) for delivery to \(v\)
  – The sender can inject only one message at a time

• Goal: find a broadcast schedule: set of point-to-point communication operations performed in a certain order
• Flat-tree: roots sends directly to all processes

• Binomial tree (MPICH)

• Multi-level tree with each level representing a different type of communication (MPICH-G2):
  – Top level is slowest (wide area networking)
  – Bottom level is fastest (parallel machine/cluster)
  – Does not prescribe how to do broadcast within a level
  – A level can include a large number of nodes, e.g., machines at various campuses

• Single-source shortest path combined with a labeling algorithm to find the schedule
Single Source Shortest Path is not optimal for broadcast: $1 + 5\Delta$

Broadcast schedule with time $2 + 3\Delta$, better for $\Delta > 0.5$. 
Binomial Tree (used by MPICH)

Gives good results when the completion time of a send is close to the completion time of the matching receive.

By contrast, flat tree is good when the completion time of a send is much smaller than that of the receive.
Proposed Method

- Find a tree $T$ that represents the communication topology
  - vertex (machine) receives the message from the parent vertex
  - single-source shortest path combined with a labeling algorithm to find the schedule
  - extend single source shortest path to incorporate the effect of the injection time

- Determine the order of sending the messages along the edges of the tree, in terms of a vertex labeling
Communication tree: extend Dijkstra’s single-source shortest path to account for the injection time

When updating the distance to $v$,
- change the distance comparison from

$$\text{dist}[v] > \text{dist}[u] + w(u,v)$$

to

$$\text{dist}[v] > \text{dist}[u] + w(u,v) + \Delta(u,v)$$

- If $\text{dist}[v]$ is updated, increase $d[u]$ with the injection time

$$\text{dist}[u] = \text{dist}[u] + \Delta(u,v)$$
dist[0:V-1] = infinity; // V = number of vertices
dist[root] = 0; // dist = length of path from root
parent[root] = NULL; // parent defines the E-SSSP tree
queue = init_queue(G, dist); // priority queue by dist
while ( ( u = dequeue_min(queue) ) ) {
    // shortest distance from u to all neighbors in queue
    while ( (v = get_next_neighbor(G, u)) ) {
        if(v ∈ queue && dist[v] > dist[u] + w(u,v) + Δ(u,v) ) {

// new min for dist[v]
dist[v] = dist[u] + w(u,v) + \Delta(u,v);
decrease_key( queue, v, dist[v] );

// add the injection time to dist[u]
dist[u] = dist[u] + \Delta(u,v);

increase_key( queue, u, dist[u] );

parent[v] = u; // update E-SSSP tree
}

} // end get_next_neighbor

} // end dequeue_min
Vertex Labeling

• Label the vertices: the label of a vertex $u$ is the time it takes for the messages sent from $u$ to reach all the vertices in the subtree rooted at $u$.

• Label vertices recursively

  $$\text{Label}(u) = 0, \text{ if } u \text{ is a leaf}$$

  $$\max\{ \text{label}(v_i) + w(u, v_i) + i \Delta(u, v_i), v \in E \}$$

  $$\text{u is not a leaf}$$

  where the vertices $v_i$ are arranged such that

  $$\text{label}(v_1) + w(u, v_1) \geq \text{label}(v_2) + w(u, v_2) \geq \ldots$$

• The label of $u$ is the smallest label given the labels of the children.
• Label the vertices: the label of a vertex $u$ is the time it takes for the messages sent from $u$ to reach all the vertices in the subtree rooted at $u$

• Label vertices recursively

$$\text{Label}(u) = \begin{cases} 0, & \text{if } u \text{ is a leaf} \\ \max \{ \text{label}(v_i) + w(u, v_i) + i \Delta(u, v_i), v \in E \} & \text{u is not a leaf} \end{cases}$$

where the vertices $v_i$ are arranged such that

$$\text{label}(v_1) + w(u, v_1) \geq \text{label}(v_2) + w(u, v_2) \geq \ldots$$

• The label of $u$ is the smallest label given the labels of the children
label_nodes( T, V, E, w, u, label) {
    // T is the E-SSSP tree
    label[u] = 0;
    if ( u is a leaf in T ) return;
    children = adjacency_list(T, u);
    // recursion
    while ( v = get_next_vertex(children) ) {
        label_nodes( T, V, E, w, v, label);
    }
}
// sort by decreasing label(v_i) + w(u, vi)
sort_decreasing_label(children, label, w);
count = 1;

// find label[u];
label[u] = 0;
while ( v = get_next_vertex(children) ) {
    if ( label[u] < label[v] + w(u,v) + count*Δ(u,v) ) {
        label[u] = label[v] + w(u,v) + count*Δ(u,v) ;
    }
    count++;
}

Implementation of Broadcast 1

extended_single_source_shortest_path();
label_nodes(); // children sorted by decreasing label
src = get_parent();
clist = get_children();
if (src != NULL) {
    MPI_Recv(src);
}
foreach dest in clist {
    { MPI_Send(dest); }
}
foreach dest in clist {
    { MPI_Recv(src); }
    if (src != NULL) {
        clist = get_children();
        src = get_parent();
        label_nodes(); // children sorted by decreasing label
        extended_single_source_shortest_path();
    }
}
Implementation of Broadcast 2

- MPI_Send() rather than MPI_Isend()
- Safe use of MPI_Isend requires, in addition to invoking either MPI_Waitall or MPI_test, some form of handshaking between sender and receiver, e.g., receiver sends a “ready to receive” message
  - MPI_Isend uses message buffers and the handshake makes sure that the buffers are not overrun
  - But handshake introduces additional synchronization overhead
Measuring the latency

\[ \lambda = \frac{(e - s - (\Delta_1 + \Delta_2))}{2} \]
Testbed

- Eight machines (Pentium II, III, and 4) located at two NRC campuses in Ottawa (about 6 miles apart)
- Injection times and latencies between nodes span a significant interval

\[ \lambda = 0.085 \ldots 1.2 \text{ ms} \]
\[ \Delta = 0.035 \ldots 0.3 \text{ ms} \]
Broadcast time vs Number of processors

The graph shows the broadcast time in milliseconds (msec) as a function of the number of processors. It compares different methods:

- **MPI_Bcast (2k)**: Dotted blue line.
- **MPI_Bcast (8k)**: Solid blue line.
- **proposed method (2k)**: Dashed red line.
- **proposed method (8k)**: Solid red line.

The x-axis represents the number of processors, ranging from 5 to 13, while the y-axis represents the time in milliseconds, ranging from 0.5 to 4.5.
Broadcast time vs Message Size

- **MPI_Bcast (13 procs)**
- **MPI_Bcast (7 procs)**
- **proposed method (13 procs)**
- **proposed method (7 procs)**

The diagram shows the relationship between broadcast time (in milliseconds) and message size (in KB) for different methods and configurations.
Conclusions

- Proposed method outperforms MPICH for small and moderate message sizes.
- For large message sizes, the injection time includes the effect of bandwidth if MPI_Send is used for point-to-point communication, and the model becomes inaccurate.
- Future work
  - replace MPI_Send with MPI_Isend, to improve performance and model accuracy
  - Include the bandwidth in the model